BASEMENT

MASSACHUSETTS
INSTITUTE of TECHNOLOGY
LIBRARIES

MASS INST TECH.
DEC 21 1989
LIBRARIES

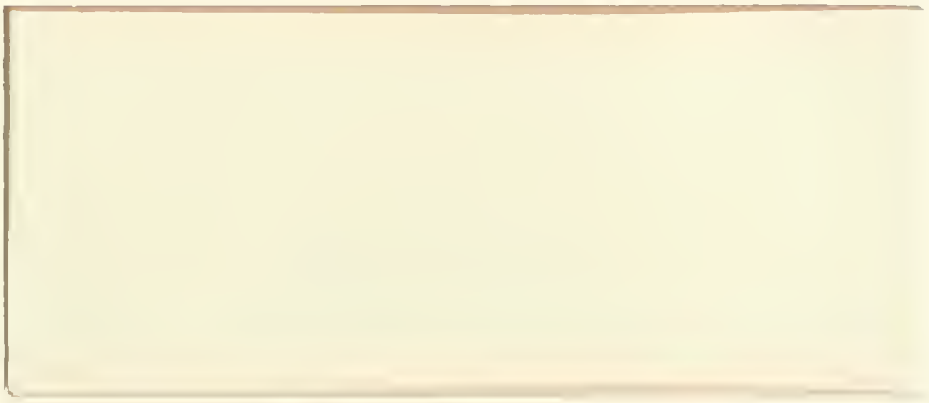
WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

Market-like Task Scheduling in Distributed Computing Environments

Thomas W. Malone
Richard E. Fikes
Kenneth R. Grant
Michael T. Howard

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

HD28
.M41



Market-like Task Scheduling in Distributed Computing Environments

Thomas W. Malone
Richard E. Fikes
Kenneth R. Grant
Michael T. Howard

March 1986

CISR WP No. 139
Sloan WP No. 1785-86

© 1986 T.W. Malone, R.E. Fikes, K.R. Grant, M.T. Howard

Center For Information Systems Research
Sloan School of Management
Massachusetts Institute of Technology

DEC 2 1 1989

Abstract

This paper focuses on a class of market-like methods for decentralized scheduling of tasks in distributed computing networks. In these methods, processors send out "requests for bids" on tasks to be done and other processors respond with "bids" giving estimated completion times that can reflect such factors as machine speed and data availability. A simple and general protocol for such scheduling is described and simulated in a wide variety of situations (e.g., different network configurations, system loads, and message delay times). The protocol is found to provide substantial performance improvements over processing tasks on the machines at which they originate even in the face of relatively large message delays and relatively innaccurate estimates of processing times. The protocol also performs well in comparison to one simpler and one more complex alternative. In the final section of the paper, a prototype system is described that uses this protocol for sharing tasks among personal workstations on a local area network.

Market-like Task Scheduling in Distributed Computing Environments

With the rapid spread of personal computer networks and the increasing availability of low cost VLSI processors, the opportunities for massive use of parallel and distributed computing are becoming more and more compelling ([Jon80]; [Gaj85]; [Dav 81b]; [Ens81]; [Ber82]; [Bir82]). One of the fundamental problems that must be solved by all such systems is the problem of how to schedule tasks on processors. This problem is, of course, a well-known one in traditional operating systems and scheduling theory, and there are a number of mathematical and software techniques for solving it in both single and multi-processor systems (e.g.,[Bri73]; [Cof73]; [Con67]; [Jon80]; [Kri71]; [Lam68]; [Kle81]; [Wit80]; [Vnt81]).

Almost all the traditional work in this area, however, deals with centralized scheduling techniques, where all the information is brought to one place and the decisions are made there. In highly parallel systems where the information used in scheduling and the resulting actions are distributed over a number of different processors, there may be substantial benefits from developing decentralized scheduling techniques. For example, when a centralized scheduler fails, the entire system is brought to a halt, but systems that use decentralized scheduling techniques can continue to operate with all the remaining nodes. Furthermore, much of the information used in scheduling is inherently distributed and rapidly changing (e.g , momentary system load). Thus, decentralized scheduling techniques can "bring the decisions to the information" rather than having to constantly transmit the information to a centralized decision maker. Because of these advantages, a growing body of recent work has begun to explore such decentralized scheduling techniques in more detail (e.g., [Sta85]; [Sin85]; [Liv82]; [Mal83]; [Lar82]; [Smi80]; [Sto77]; [Sto78]; [Cho79]; [Bry81]; [Sta84]; [Ten81a]; [Ten81b]; [Str81] ; [SRC85]; see Stankovic for a useful review).

In this paper, we focus on a particular class of decentralized scheduling techniques: those involving market-like "bidding" mechanisms to assign tasks to processors (e.g., [Mal83]; [Far72]; [Smi80]; [Sin85]; [Far73]; [Sta84]). Such techniques are remarkably flexible in terms of the kinds of factors they can take into account: job characteristics, processor capacities and speeds, current network loading and current locations of data and related tasks (e.g., see [Sta84]). In succeeding sections of the paper, we will describe a simple but powerful bidding protocol and present detailed simulation and analytic results that explore the behavior of the protocol in a wide variety of stituations. These results apply to many forms of parallel computation, regardless of whether or not the processors are geographically separated and whether or not they share memory.

Motivating example

Even though the simulation results are applicable in many situations, the driving example that motivated the development and analysis of our protocol was the increasingly common situation of large numbers of personal workstations connected by local area networks (e.g., [Bir82]; [Bog80]). In the final section of the paper, we describe a prototype system, called Enterprise, that uses the protocol to share tasks among workstations in such a network.

One of the benefits of sharing tasks in such networks is that a new philosophy for designing distributed systems becomes possible. The traditional philosophy used in designing distributed systems based on local area networks is to have dedicated personal workstations which remain idle when not used by their owners, and dedicated special purpose servers such as file servers, print servers, and various kinds of data base servers (e.g., [Bir82]; [Sch84]). A system like Enterprise that schedules tasks on the best processor available at run time (either remote or local) enables a much more flexible design. In this new philosophy, personal workstations are still dedicated to their owners, but during the (often substantial) periods of the day when their owners are not using them, these personal workstations become general purpose servers, available to other users on the network. "Server" functions can migrate and replicate as needed on otherwise unused machines (except for those such as file servers and print servers that are required to run on specific machines). Thus programs can be written to take advantage of the maximum amount of processing power and parallelism available on a network at any time, with little extra cost when there are few extra machines available.

Problem description and related work

In describing the problem on which we are focusing, it is useful, first of all, to distinguish between two components of task scheduling in distributed networks: (1) the *assignment* of tasks to processors, and (2) the *sequencing* of task once they have been assigned to processors. Much previous work on distributed load sharing has focused only on the task assignment problem (e.g., [Liv82]; [Sta85]; [Cho79]) and used first come-first served (FCFS) sequencing. It is clear, however, that task sequencing can have a major effect on system performance measures (e.g., see [Con67]). The task scheduling method we will describe solves both the task assignment and the task sequencing problems simultaneously.

Another common simplifying assumption in much previous work (e.g., [Liv82]; [Bry81]) is that all processors in the scheduling network are identiçal. It is frequently the case, however, in our motivating example of workstations on networks and in many other distributed processing situations, that there are important differences between processors. These differences include factors such as speed, ability to do certain tasks at all, and whether the necessary the data or programs are already present on the processor or must be transmitted from elsewhere on the network. Our scheduling method is designed to accomodate a wide variety of such processor differences.

In some cases (e.g., [Bry81]; [Sta84]), it is desirable to be able to have more than one task on a processor at the same time and to move tasks from one processor to another after the tasks have begun execution. These capabilities often add substantially to the complexity of implementing a real system, however, so we have simplified our analysis by omitting them.

The scheduling methods we will consider can be applied in any situation that has the properties we have just described: tasks are sequenced one at a time on heterogeneous processors without being moved or preempted after execution begins.

Bidding mechanisms

We mentioned above a number of advantages (e.g., reliability and flexibility) of bidding mechanisms for task scheduling in distributed networks. We define a *bidding mechanism* for distributed task scheduling to be one in which (1) task descriptions are broadcast (by "clients") to a set of possible processors ("contractors"), (2) some subset of the contractors respond with "bids" indicating their availability (and possibly cost) for performing the task, and (3) one of the bidders is selected to perform the task.

Note that there are many characteristics of human markets that are not included in this definition. For example, bidding mechanisms as we have defined them here can assign tasks after only one "round" of bid submissions. They need not include any of the iterative price adjustment, based on supply and demand, that is widely discussed in microeconomic theory (e.g., [Arr71]), and that is included in the network channel access scheduler described by Kurose, Schwartz, and Yemini [Kur85]. In fact, our early experiments with an iterative pricing mechanism for distributed task assignment [Mal82] discouraged us from pursuing this approach further because of the difficulties with guaranteeing convergence at all ([Arr60]; [Arr71]) and because of the extremely computationally intensive nature of the iterative process.

Several of the systems mentioned above ([Far72]; [Sta84]; [Ram84]) use non-iterative bidding mechanisms, but for scheduling problems different from the one we are considering. One previous system, however, can be used for problems of the type in which we are interested: The contract net protocol ([Smi80]; [Smi81]; [Dav83]) is a very general protocol for distributing tasks in a network of heterogenous processors based on whatever task specific criteria are provided by the system designer. Its "announcement, bid, award" sequence allows for mutual selection of clients and contractors; that is, contractors choose which clients to serve and clients choose which contractors to use. A later system ([Sin85]) which is based in part on the contract net protocol and on an earlier report of the system described here ([Mal83]), shows how the protocol can be modified to substantially reduce the number of messages required.

In order to achieve any particular scheduling objective with these systems, however, specific selection criteria must be developed. The most important way in which our scheduling protocol differs from the contract net protocol is by specializing the selection criteria to two primary dimensions: (1) contractors select clients' tasks in the order of numerical *task priorities*, and (2) clients select contractors on the basis of *estimated completion times* (from among the contractors that satisfy the minimum requirements to perform the job). We will see below how this simple specialization of the generalized contract net protocol allows us to make direct use of a variety of optimality results from traditional scheduling theory, including those about minimizing mean flow times, and to do detailed evaluations of the effect of numerous factors on scheduling performance.

One of the potential problems that arises in bidding systems like the contract net protocol stems from the fact that only idle processors submit bids. Thus assignment decisions are made in the absence of any information about the loads and capabilities of processors that are busy at the time the bids are evaluated. This may be undesirable, for example, in situations where only slow processors are available at the time the task arrives, but a much faster processor will become available soon. In the sections below, we consider two alternative solutions to this problem. Our primary scheduling method, called the Distributed Scheduling Protocol, uses the simple technique of canceling and restarting tasks if the later bid is sufficiently better. We also consider, in a later section, a much more elaborate alternative method in which processors maintain detailed future schedules of tasks they have agreed to perform and use this information in submitting their bids. Our simulation results investigate the performance of both these methods in a variety of situations.

THE DISTRIBUTED SCHEDULING PROTOCOL

Figure 1 illustrates the steps in our primary scheduling process, the Distributed Scheduling Protocol (DSP):

1. *The client broadcasts a "request for bids".* The request for bids includes the priority of the task, any special requirements, and a summary description of the task that allows contractors to estimate its processing time.

2. *Idle contractors respond with "bids" giving their estimated completion times. Busy contractors respond with "acknowledgements" and add the task to their queues (in order of priority).*

3. *When a contractor becomes idle, it submits a bid for the next task on its queue.*

4. *(a) If more than one bid has been received when the client evaluates bids, the task is sent to the best bidder.* The length of time to wait before evaluating bids is a parameter that is set depending on the message delay time.

   (b) If no bids have been received when the client evaluates bids, the task is sent to the first subsequent bidder.

   (c) If a later bid is "significantly better" than the early one, the client cancels the task on the first bidder and sends the task to the later bidder. The criterion for deciding whether a late bid is "significantly better" is a parameter, the effect of which is examined in the simulations below. If the later bid is not significantly better (or if the task has side-effects and cannot be restarted), the client sends a cancel message to the later bidder.

5. *When a contractor finishes a task, it returns the result to the client.*

6. *When a client receives the result of a task, it broadcasts a "cancel" message so that all the contractors can remove the task from their queues.*

A more detailed description of the DSP, including complete specifications of the message contents, is provided by Malone, Fikes, and Howard ([Mal83]).

Global Scheduling Objectives

One of the advantages of this protocol is that it separates the *policy* decisions about how priorities are assigned from the *mechanism* of actually scheduling tasks according to these priorities (e.g., [Cof73], [Lam68]). Traditional schedulers for centralized computing systems often use *list scheduling* as a basis for layering the design of a system (e.g., [Cof73]). In this approach, one level of the system sequences jobs according to their order in a priority list while the policy decisions about how priorities are assigned are made at a higher level in the system (see [Lam68]). DSP allows precisely the same kind of separation of policy and mechanism. The DSP protocol itself is concerned only with sequencing jobs according to priorities assigned at some higher level. By assigning these priorities in different ways, the designers of distributed systems can achieve different global objectives. For example, it is well known that in systems of identical processors, the average waiting time of jobs is minimized by doing the shortest jobs first [Con67]. Thus, by assigning priorities in order of job length, the completely decentralized decisions based on priority result in a globally optimal sequencing of tasks on processors.

Optimality results for mean flow time and maximum flow time. Traditional scheduling theory (e.g., [Con67]) has been primarily concerned with minimizing one of two objectives: (1) the average flow time of jobs (F_{ave})--the average time from availability of a job until it is completed, and (2) the maximum flow time of jobs (F_{max})--the time until the completion of the last job. Minimizing F_{max} also maximizes the utilization of the processors being scheduled [Cof73]. (A third class of results from scheduling theory, involving the "tardiness" of jobs in relation to their respective deadlines, appears to be less useful in most computer system scheduling problems.) The most general forms of both these problems are NP-complete ([Bru74], [Iba77]), so much of the literature in this field has involved comparing scheduling heuristics in terms of bounds on computational complexity and "goodness" of the resulting schedules relative to optimal schedules (e.g , [Dav81a], [Jaf80]).

A number of results suggest the value of using two simple heuristics, shortest processing time first (SPT) and longest processing time first (LPT), to achieve the objectives F_{ave} and F_{max}, respectively. First, we consider cases where all jobs are available at the same time and their processing times are known exactly. In these cases, if all the processors are identical, then SPT exactly minimizes F_{ave} [Con67] and LPT is guaranteed to produce an F_{max} that is no worse than 4/3 of the minimum possible value [Gra69]. If some processors are uniformly faster than others, then the LPT heuristic is guaranteed to produce an F_{max} no worse than twice the best possible value [Gon77]. Next, we consider cases where all jobs are available at the same time but their exact processing times are not known in advance. Instead the processing times have certain random distributions (e.g., exponential) with different expected values for different jobs. In these cases, if the system contains identical

processors on which preemptions and sharing are allowed, then SPT and LPT exactly minimize the expected values of F_{ave} and F_{max}, respectively, ([Web82], [Gla79]). Finally, we consider cases where the jobs are not all available at the same time but instead arrive randomly and have exponentially distributed processing times. In these cases, if the processors are identical and allow preemption, then LPT exactly minimizes F_{max} [Van81].

Other scheduling objectives. DSP can also be used to achieve many other possible objectives besides the traditional ones of minimizing mean or maximum flow time for independent jobs. For example:

(1) *Parallel heuristic search.* Many artificial intelligence programs use various kinds of heuristics for determining which of several alternatives in a search space to explore next. For example, in a traditional "best first" heuristic search, the single most promising alternative at each point is always chosen to be explored next [Nil80]. By using the heuristic evaluation function to determine priorities for DSP, a system with n processors available can be always exploring the n most promising alternatives rather than only one. Furthermore, if the processors have different capabilities, each task will be executing on the best processor available to it, given its priority.

The DSP protocol can also, of course, be used in heuristic searches in a more straightforward way to assign a fixed set of subtasks to processors. For example, Singh and Genesereth's ([Sin85]) system uses a bidding protocol to assign deduction steps to distributed processors. Their use of the protocol appears to be minimizing F_{max} by giving highest priority to the most costly tasks.

(2) *Arbitrary market with priority points.* Another obvious use of DSP is to assign each human user of the system a fixed number of priority points in each time period. Users (or their programs) can then allocate these priority points to tasks in any way they choose in order to obtain the response times they desire (see [Sut68] for a similar--though non-automated--scheme, and Mendelson ([Men85]) for a related analysis aimed at determining, not micro-level priorities, but macro-level chargeback policies).

(3) *Incentive market with priority points.* If the personal computers on a network are assigned to different people, then a slight modification of the arbitrary market in (2) can be used to give people an incentive to make their personal computers available as contractors. In this modified scheme, people accumulate additional priority points for their own later use, every time their machine acts as a contractor for someone else's task.

Estimating processing time

As described above, DSP requires bidders to estimate their completion times for different tasks. This information is used by DSP only to rank different tasks and different contractors, so only rough estimates are needed. In some cases, historical processing times for similar jobs might provide a basis for making even more precise estimates, possibly using parameters such as the size of the input files. Our simulation studies below include an examination of the consequences of making these estimates very poorly.

Alternative Scheduling Protocols

For comparison purposes, we now consider two alternative protocols. The first protocol is a scheme designed to remedy one of the possible deficiencies of DSP. The second is a random assignment method that provides a comparison with designs where no attention is given to the scheduling decision.

Alternative 1 - Eager assignment

As discussed above, one of the possible deficiencies of DSP is that no estimates of completion times are provided by processors that are not ready to start immediately. That is, clients using DSP may start a task on a machine that is available immediately (possbily their own local machine), only to find that another much faster machine becomes available soon. If the task is canceled and restarted, all the processing time on the first machine is wasted. If not, the job finishes later than it could have. If reasonable estimates of completion times on currently busy machines are available, then clients would know enough to wait for faster machines that were not immediately available.

In this alternative scheduling protocol, tasks are assigned to contractors as soon as possible after the tasks become available and then reassigned as necessary when conditions change. In this way, each contractor maintains a schedule of tasks it is expected to do, along with their estimated start and finish times, and so the contractor can make estimates of when it could complete any new task that is submitted. By analogy with "lazy" evaluation of variables ([Fri76], [Hen76]) the original DSP could be called "lazy assignment" because clients defer assigning a task to a specific contractor until the contractor is actually ready to start. This alternative protocol, therefore, will be called "eager assignment," since it assigns tasks to contractors as soon as possible. This alternative may be thought of as a logical extension of the "immediate bid responses" used in the contract nets protocol

([Smi80], [Smi81]), and is also similar to the "STB" scheduling alternative considered by Livny & Melman ([Liv82]).

In this protocol, all contractors bid on all tasks even if they are currently busy. A contractor estimates its starting time for a task by finding the first time in its schedule at which no task of higher priority is scheduled. Then the client picks the best bid and sends the task to the contractor who submitted it. When new tasks are added to a contractor's schedule, or when a task takes longer than expected to complete, the contractor notifies the owners of later tasks in its schedule that their reservations have been "bumped." These clients may then try to reschedule their tasks on other contractors. A related kind of bumping process occurs in the Distributed Computing System ([Far73]; [Far 73]) when a processor that has bid on a task is no longer available by the time the task arrives.

It is important to note that even in cases where there is a lot of bumping, this scheduling process is guaranteed to converge. Since tasks can only bump the reservations of other tasks of lower priority, the scheduling of a new task can never cause more than a finite number of bumps. To reduce the finite (but possibly large) amount of rescheduling in rapidly changing situations, bumping occurs only when the currently estimated completion time of job exceeds its original estimate by the amount specified in the "bump tolerance" parameter.

While this alternative is clearly more elaborate and may require much more message traffic than the DSP, it may also result in better schedules for situations with processors of widely varying capabilities.

Alternative 2 - Random assignment

In the second alternative protocol, clients pick the first contractor who responds to their request for bids and contractors pick the first tasks they receive after an idle period. Contractors do not bid at all when they are executing a task, and they answer all requests for bids when they are idle. If a client does not receive any bids, it continues to rebroadcast the request for bids periodically. When contractors receive a task after already beginning execution of another one, the new task is rejected (with a "bump" message) and the client who submitted it continues trying to schedule it elsewhere. In the simulations discussed below, the selection of the first bidders when more than one machine is available, and of the first task when more than one task is waiting, are both modeled as random choices since the delay times for message transmission and processing are presumably random. (In reality, fast contractor machines might often respond more quickly to requests for bids than slow ones

and so would be more likely to be the first bidders. Thus the performance of this scheduling mechanism in a real system might be somewhat better than the simulated performance.)

SIMULATION RESULTS

In many real distributed scheduling environments, including our motivating example of workstations on a network, minimizing the mean flow time of independent jobs is likely to be the primary scheduling objective. Unfortunately, as we noted above, the problem of scheduling tasks to meet this objective is usually NP-hard, and other analytic results about the effects of sequencing strategies (other than random) are quite scarce. It is therefore appropriate to rely heavily on simulations to investigate strategies for achieving this objective. In this section, we summarize the results of a series of simulation studies that investigate the performance of the DSP in a variety of situations and compare it to the two alternatives described above (eager assignment and random assignment). We also report several analytic results that are useful for comparisons to the simulation results.

Simulation Method

Priorities. Since the objective to be minimized by scheduling is assumed to be the mean flow time of jobs, priorities in all simulations were determined according to the shortest processing time first (SPT) heuristic, except in the random alternative where priorities are not used.

Network configurations. Ten different configurations of machines on the network were defined. In all configurations, a total of 8 units of processing power was available, but in different cases this was achieved in different ways: a single machine of speed 8; or 8 machines of speed 1; or 1 machine of speed 4 and 2 machines of speed 2; etc. We will denote different network configurations below by a sequence of the machine speeds they contain (e.g., "422").

Job loads. For all the simulations, jobs were assumed to be independent of each other and suitable for processing on any machine in the network. The job arrivals were assumed to be a Poisson process and the amount of processing in each job was assumed to be exponentially distributed (with a mean of 60 time units on a processor of speed 1). System utilization was defined as the expected amount of processing requested per time interval divided by the total amount of processing power in the system, and three different levels of system utilization (0.1, 0.5, and 0.9) were simulated.

In order to increase the power of comparisons between different simulations at the same utilization level, the variance reduction technique called *common random numbers* was used (see [Law82a], pp. 350-354). In this technique, the same sequence of random numbers is used to generate jobs in each of the different simulations. By reducing the variation between simulations that is due to random job generation, this technique increases the statistical power of the comparisons between the factors of interest (different network configurations and scheduling methods).

Statistical tests for equilibrium. A generic problem in simulation studies is determining how many jobs to simulate. In all the simulations reported below, the procedure developed by Law and Carson ([Law79]) was used to determine when to terminate the simulation and what confidence intervals to report for the steady state mean. This procedure was recommended by Law and Kelton ([Law82a, Law82b]) after comprehensive surveys of both fixed-sample-size and sequential procedures. The procedure is based on the idea of comparing successive batches of jobs to determine whether the means are correlated. When the number of batches and the batch sizes are large enough for the batch means to be uncorrelated, then the grand mean and variance are unbiased estimates of the steady state values for the simulation (see [Law79]).

In practice, this test was quite stringent. Each simulation was run until a 90 percent confidence interval could be computed with a width of less than 15 percent of the estimated mean. The number of jobs required ranged from 1200 jobs in some of the simulations for system utilization of 0.1 to over 75,000 jobs in some of the simulations for system utilization of 0.9.

In many simulation studies, some number of jobs are discarded from the beginning of the analysis to remove the "start up" values from the overall mean. However, as Law and Kelton ([Law82b]) and Gafarian, Ancker, and Morisaku ([Gaf78]) have noted, there are no generally satisfactory methods for determining how many jobs to discard. Therefore, we adopted the conservative approach of not discarding any jobs. Since, if anything, this increases the variance between the means of early and later batches, it may increase the number of jobs necessary to pass the steady state test described above, but it will not result in biased estimates of the steady state mean (see [Law82a]).

Accuracy of job processing time estimates. In addition to the actual amount of processing in each job, the jobs also included an estimated amount of processing for each job (i.e., the estimate a user might have made of how long the job would take). These estimates are used to determine job priorities and estimated completion times. In order to examine extreme cases, these estimates were either perfect (0

percent error) or relatively inaccurate (+/- 100 percent error). In the case of inaccurate estimates, the errors were uniformly randomly distributed over the range.

Communications delays. In order to simulate "pure" cases of the different scheduling mechanisms, most of our simulations treat communication among machines as perfectly reliable and instantaneous. In real situations where communication delays are negligible relative to job processing times, this assumption of instantaneous communications is appropriate.

In other simulations, designed to explore the effect of communication delays, we assumed constant delays for the transmission of all messages. The values for message delay that were simulated were equivalent to 0%, 5%, 10%, and 15% respectively of the average job processing time. The simulations include only the effect of message delays; they do not include any other factors such as processing overhead needed to transmit and receive messages.

Even though, as we will see below, the effect of increasing communication delays is quite linear, the results cannot be obtained by simply adding the total message delay time per job (the delay required for 4 messages) to the results for no delays. This simple approach does not work because it neglects the time saved by having the announcements of waiting jobs already queued at processors that are busy when the jobs are first announced.

Restarting after late bids. In most of the simulations of DSP, late bids are never accepted no matter how much of an improvement they are over the earlier bids. In one series of simulations, designed to test the effect of this parameter, a range of values for this "late bid improvement" parameter is investigated.

Bumping and rebroadcasting. In keeping with the spirit of simulating "pure" scheduling methods, jobs in the eager simulations are rescheduled every time their scheduled start time is delayed at all. In a real system, jobs would ordinarily have to be bumped by more than some tolerance before being rescheduled. In other words, the performance of the eager method could only get worse if fewer bumps were made.

Similarly, in the random assignment simulations, clients rebroadcast their requests for bids in every time interval of the simulation until the job is successfully assigned to a contractor. Thus, this simulates the best scheduling performance the random method could achieve; if rebroadcasting occurred less often, the performance could only get worse.

Analytic methods

It is possible to derive analytically several simple results that can be used for comparison with the simulation results.

Random assignment and sequencing on identical processors. The case where tasks are assigned randomly to identical processors as they become available and sequenced randomly on these processors is equivalent to an M/M/m queuing system, that is, a system with m servers for one queue. The expected queue length and waiting time for such a system are, respectively:

$$L = [(m\lambda/\mu)^m(\lambda/\mu)P_0] \,/\, [m!(1-\lambda/\mu)^2] \,,$$

$$W = L/m\lambda + 1/\mu \,,$$

where $P_0 = 1/\left[\sum_{i=0}^{m-1} (m\lambda/\mu)^i/i! + [(m\lambda/\mu)^m/m!]/1-\lambda/\mu) \right]$.

Our simulations do, in fact, generate confidence intervals that include these values for the cases where they can be computed. In these cases, only the analytically derived results are reported.

Optimal assignment (with jobs moving during execution and with random sequencing). Even though our simulations assume that jobs cannot be moved once they have begun execution there is a simple formula for the flow times that would result if jobs could be moved during execution in such a way that the fastest processors were always in use (see [Sta85]). This formula provides a lower bound for the results of optimal assignment. However, it assumes that tasks are sequenced randomly on the machines to which they are assigned. Since our simulated scheduling methods include sequencing as well as assignment, they might do better or worse than these analytic results depending on the relative importance of sequencing and assignment in a given situation. They do, however, provide an approximate lower bound for rough comparison purposes.

In order to obtain these results, we assume that the processor service rates μ_i are ordered so that $\mu_i \geq \mu_j$ if $i \leq j$. Letting the number of processors be N, we define

$$\mu(i) = \sum_{j=1}^{i} \mu_j \quad ,$$

$$M(i) = \prod_{j=1}^{i} \mu(i) \quad i = 1,2,\ldots,$$

$$M(0) = 1.$$

Using straightforward assumptions about the state transition probabilities and the conservation of flow principle, Stankovic ([Sta85]) shows that the expected queue length and waiting time for such a system are, respectively:

$$W = P_0\left(\sum_{j=1}^{N-1} j\frac{\lambda^j}{M(j)} + \frac{\lambda^N}{M(N)}\left[\frac{N-1}{1-\lambda/\mu(N)} + (1-\lambda/\mu(N))^{-2} \right] \right) / \lambda \ ,$$

$$\textit{where } P_0 = 1 \ / \left(\sum_{j=0}^{N-1} \lambda^j/M(j) + \frac{\lambda^N}{M(N)} \ / \ (1-\lambda/\mu(N)) \right).$$

Local processing only. One question of interest in our study is the amount of speedup in mean flow time that is possible from scheduling jobs anywhere on the network as opposed to processing all jobs locally on the machine at which they originate. As noted above, analytic results are not available for the effect of sequencing in this situation. However, it is possible to use the simulation results for the case with only one machine in the network to estimate of the effect of sequencing and then compute the overall flow times that would result from all machines performing their local scheduling similarly.

To do this, we take advantage of the fact that the time units used in the simulation are arbitrary and can be scaled as desired. For example, the mean flow time for a single machine of speed 1 is 8 times that of a single machine of speed 8, since the two simulations are indistinguishable except for the time units. In general, let n_i be the number of machines of type i, s_i be the speed of the machines of type i, $C = \Sigma_i s_i n_i$ be the total processing capacity on the network, and W_1 be the mean flow time in a network containing only a single machine of speed 1. Then the the average flow time in the total network is

$$W = \Sigma_i (n_i/C)W_1.$$

Results

Relative effects of system load, network configuration, and accuracy of processing time estimates

We first investigate the results of DSP scheduling with a variety of system loads, network configurations, and accuracies of processing time estimates. In order to focus on the effect of these factors, message delays were kept constant (at 0), and late bids were never accepted, no matter how much better they were (i.e., the "late bid improvement" parameter was effectively infinite). Table 1 lists these results, Figure 2 shows the effect of system load for a typical configuration (1 machine of speed 4 and 4 machines of speed 1), and Figure 3 shows the effect of network configuration holding constant both system load (at 50% utilization) and accuracy of processing time estimates (at 0% error).

Both system load and network configuration have a major impact on mean flow time (as much as a factor of 4 for the range of conditions we studied), while the accuracy of processing time estimates does not appear to be a major factor in performance (at most a difference of about 15% when estimates have errors of up to 100%).

Figure 3 shows that the effect of dividing the same amount of processing power into more and more processors is almost linear and that the number of processors used has a much greater impact on flow time than the maximum range of processor speeds. Even if we restrict our attention to the first part of the graph where the number of processors ranges from 1 to 4, we see that this makes a difference of approximately a factor of 3 in flow time, while changing from a configuration with identical processors to one with a speed range of a factor of 4 makes only about a 12 percent difference in flow time.

Comparing the results in Table 1 for DSP with the analytically derived results for optimal assignment (with moving during execution and random sequencing), we see that in those cases where assignment is important (i.e., where processor speed ranges are large), DSP does reasonably well in comparison to this rough "lower bound". In the cases where assignment makes no difference (i.e., where processor speeds are identical), DSP does much better than the rough "lower bound" because DSP does intelligent sequencing as well as assignment.

Increasing the size of the bidding network while keeping overall utilization constant

The results we have just seen all involve configurations in which the total amount of processing power in the network is constant (a total of 8 processing units), but divided among processors in different ways. From the point of view of a network designer, another relevant question is how many processors to combine in one bidding network, that is, how many processors to group together for the purpose of sharing tasks. To answer this question, we assume that the speed of each processor and the overall utilization remain constant and then consider the effect of adding processors to the network.

We can estimate this effect in two ways. The first, purely analytic, method uses the well-known formulas given above for random scheduling on identical machines (i.e., both random assignment and random sequencing). Since the machines are identical, random assignment is as good as any assignment method, but random sequencing is certainly not optimal (e.g., see [Con67]). Since simple ways of analytically computing the effects of sequencing are not known, the second method uses the results of the DSP simulations to estimate the effect of (shortest processing time first) sequencing. These estimates are then adjusted analytically to refer to networks with different numbers of machines, all of the same speed (speed 1). To make this adjustment, we scale time units as described above. If $W_{N,s}$ is the mean flow time for a network with N identical machines of speed s, then a network with N machines of speed s' would have a mean flow time of $W_{N,s'} = (s/s')W_{N,s}$.

Table 2 shows the results of both these methods, and Figure 4 plots the results for the second method. It is clear that using either random or SPT sequencing, pooling work from several processors can, indeed, have a significant impact on flow time. This effect, however, is very dependent on the system utilization. There is essentially no benefit at low utilizations; at moderate utilizations, there is very little additional benefit after about 4 machines, and even for heavy utilizations, most of the benefits have been exhausted by about 8 machines.

It is important to realize that this result is quite general. The analytic results are based on well-known formulas for random scheduling; our simulation results show that the effect holds for SPT sequencing as well. The shape of the curves shown in the figure does not depend on processor speeds or average job processing times; changes in these factors merely change the scale of the vertical axis. Our result does depend on the assumption of exponentially distributed service times, but the intuitive argument given by Sauer and Chandy [Sau81] to explain their results suggests that the benefits of pooling jobs from several processors should be even more pronounced in systems with greater coefficients of variation in service time. Their argument says that with high coefficients of variation, multiprocessor systems perform better than single processor systems because a few very long jobs can

bottleneck a single processor while in a multiprocessor system, only one of the processors is monopolized and other jobs can still be processed on the other processors. This argument also suggests that in systems with very high coefficients of variation, the benefits of adding more processors might extend further than in systems with exponential service times.

The implication of this result for system design is quite important. It suggests that there can be significant benefits from pooling work generated by different machines in a network but that there is no need to have large numbers of machines all on the same bidding network. In most situations, several separate networks of 8 - 10 processors each should perform as well, from the point of view of reducing mean flow time, as a single network including all the processors together.

Effects of message delays

One of the possible problems with pooling work from several machines (even if only a few machines are involved) is that the message delays required for scheduling and for transferring results back and forth might overwhelm the flow time benefits obtained from pooling. Figure 5 compares the results of pooling jobs by network scheduling to the results of processing all jobs locally on the machines at which they originate. The configurations simulated were chosen from our total set of configurations to represent the situation in which the least benefit would result from pooling (only two identical machines) and the situations in which the most benefit would result (the maximum number of total machines or the maximum range of processor speeds).

The results in Figure 5 show that pooling of work can, in fact, be beneficial, even when message delays are quite substantial. With moderate loads (50%) and large numbers of processors, pooled scheduling is superior to strictly local processing even when message delays exceed 20% of the average job processing time! Even in the configuration where pooling has the least benefit (two identical machines), pooled scheduling is preferred at moderate loads up to about the point where message delays exceed 5% of the average job processing time.

Effects of "late bid improvement" parameter

Figure 6 shows the effect of varying the amount of improvement required in "late" bids before jobs will be cancelled on the machines to which they were originally sent and restarted on the late bidding machine. If we let t_E be the estimated completion time in the earlier bid, t_L be the estimated completion time in the late bid, and t be the time at which the late bid is evaluated, then the

"improvement" is $i = 1 - (t_L - t_0)/(t_E - t)$. Late bids must exceed some criterion parameter i_0 before they will be accepted.

We have simulated the configuration in which this factor could make the most difference (the configuration with the maximum range of processor speeds). The most improvement possible for a single job in this situation is 0.75 (if a processor of speed 4 becomes available immediately after a bid has been awarded to a processor of speed 1). Therefore, setting i_0 at 0.75 or greater will result in no rescheduling.

As Figure 6 shows, at low utilizations performance is improved by rescheduling for any improvement at all, no matter how small, but at moderate utilizations, performance is made slightly worse by this strategy. In both cases, the optimal setting for i_0 appears to be somewhere in the range of 0.4 to 0.6. At moderate utilizations, the maximum benefit from using this parameter appears to be about 5 percent, while at low utilizations using this parameter may result in overall flow time improvements on the order of 20 to 25 percent. This result is sensible since the only cost of rescheduling is the processing time "wasted" on the first processor, and with low utilization, this processing time is plentiful anyway. As utilizations increase, the cost of "wasting" time on the first processor to which a job is assigned increases and the potential benefit from using this parameter becomes negligible.

Evaluation of alternative scheduling methods

Figure 7 shows the results of DSP and the two alternative scheduling methods "eager" and "random." In all cases, DSP is at least as good as, and in some cases, much better than the more complicated and expensive "eager" scheduling method. With perfect estimates of processing amounts, both DSP and eager assignment are consistently as good as or better than random assignment. With poor processing time estimates, DSP suffers little performance degredation, but the performance of eager degrades quite substantially--in some cases eager becomes significantly worse than random. It is particularly striking, in view of the fact that the eager method was motivated by problems arising with large numbers of processors and large speed differences among processors, that the eager method performs worst in precisely those situations. We believe that two primary factors account for this result:

(1) *"Stable world illusion."* In the eager assignment method, each job is assigned to the machine that estimates the soonest completion time. But if jobs of higher priority arrive later and are assigned to the same machine, then they will keep "bumping" the first job back to later and later times. In other words, jobs are assigned to machines on the assumption that no

more jobs will arrive (i.e., that the world will remain stable). Even though in the simulation, jobs are rescheduled every time any new jobs arrive that delay their estimated start time, by the time the job is rescheduled, it may already have missed a chance to start on another machine that could have completed it before it will now be completed.

In some of our simulations (not included here), the bids included an extra factor to correct for this effect, that is, bids included an estimate of how long the starting time of the job would be delayed by jobs that had not yet arrived, but could be expected to arrive before the job began execution. (See [Mal86] for the derivation of this correction factor.) Even though the inclusion of this correction factor did improve the performance of the eager assignment method somewhat, the changes were not substantial.

(2) *Unexpected availability*. When a job takes longer than expected, or when higher priority jobs arrive at a processor, all the clients who submitted jobs scheduled later on that processor are notified with "bump" messages and given a chance to reschedule their jobs. When a job takes less time than expected or when jobs scheduled on a processor are canceled, the processor may become available sooner than expected, but in these cases, the clients who submitted jobs that were scheduled elsewhere but who might now want to reschedule on the newly available machine are never notified. There can thus be situations where fast processors are idle while high priority jobs wait in queues on slower processors. This appears to be a serious weakness of the eager assignment method. We have specified, but not implemented, an addition to the protocol that notifies all clients of such situations and allows them to reschedule their tasks. The cost of this addition would be even greater message traffic and system complexity, and we believe it unlikely that the resulting performance would be significantly better than the much simpler lazy assignment method.

IMPLEMENTATION OF THE ENTERPRISE SYSTEM

In this section, we describe several highlights of the Enterprise system implementation. A more detailed description is provided by Malone, Fikes, and Howard ([Mal83]). The system schedules and runs processes on a local area network of high-performance personal computers. Processes are assigned to the best machine available at run-time (whether that is the machine on which the task originated or another one). The assignment takes into account two primary factors that affect estimated completion time: machine speed and currently loaded files needed for the task. A prototype version of the system is implemented in Interlisp-D and runs on the Xerox 1100 (Dolphin),

1108 (Dandelion), and 1132 (Dorado) Scientific Information Processors connected with an Ethernet. The prototype version has received limited testing, but no significant operational experience has been obtained.

As shown in Figure 8, the system is partitioned into three layers of software. The first layer provides an Inter-Process Communication (IPC) facility by which different processes, either on the same or different machines, can send messages to each other. When the different processes are on different machines, the IPC protocol uses internetwork datagrams called PUPs (see [Bog80]) to provide reliable non-duplicated delivery of messages over a "best efforts" physical transport medium such as an Ethernet [Met76]. Enterprise uses a pre-existing protocol that is highly optimized for remote procedure calls ([Bir84], [Tho83]) in which messages are passed to remote machines as procedure calls on the remote machines. The next layer of the Enterprise system is the Distributed Scheduling Protocol (DSP) which, using the IPC, assigns and sequences the task on the best available machine. Finally, the top layer is a Remote Process Mechanism, which uses both the DSP and IPC to create processes on different machines that can communicate with each other.

The implementation assumes that the owners of idle workstations voluntarily put their machines into a mode where the machines respond to requests for bids from the network. Shock and Hupp [Sho82] describe an alternative mechanism for locating and activating idle machines on a network without their owners' intervention.

Several augmentations of the DSP protocol were made to account for (1) processor or communication failures and delays, (2) the unique status of the "local" processor, and (3) human users who might try to "game" the system.

Unreliable processors and communications

In addition to the messages involved in the bidding cycle, clients periodically query the contractors to which they have sent tasks about the status of the tasks. If a contractor fails to respond to a query (or any other message in the DSP), the client assumes the contractor has failed. Failures might result from hardware or software malfunctions or from a person preempting a machine for other uses. In any case, unless the task description specifically prohibits restarting failed tasks, the client automatically reschedules the task on another machine. Similarly, if a contractor fails to receive periodic queries from one of its clients, the contractor assumes the client has failed and the contractor aborts that client's task.

Since a task can be restarted several times during its lifetime (e.g., because of processor failures or because of a late bid improvement), there can be different "incarnations" of the same process [Nel81]. Because messages can sometimes be delayed or lost, confusions might result from messages referring to earlier incarnations of a current task. To prevent such confusions, each task is assigned a task identifier that is guaranteed to be unique across time and space. In order to do this, a timestamp of the most recent "milestone event" in the life of the process is included in the task identifier. Milestone events are the sending of either a request for bids or a task message concerning the task. Both these events render obsolete all previous DSP messages concerning the task. Before responding to DSP messages about a particular task, therefore, both clients and contractors check to be sure the message concerns the most recent incarnation of the task. (These task identifiers serve the same purpose as the call identifiers used by Birrell and Nelson [Bir84]).

In view of benefits described above from using "late bid improvement" rescheduling in lightly loaded systems (as ours was), and in view of the unpredictable delays in message transmission, the Enterprise system implements a variation of the DSP protocol in which the first bid for a task is always accepted, rather than waiting any fixed time to evaluate bids. Then if a later bid is significantly better, the task is rescheduled.

The "remote or local" decision

With the variation of DSP just described, if the local machine submits bids for its own tasks (i.e., the client machine offers to be its own contractor), then the local machine will presumably always be the first bidder and will therefore receive every task. To prevent this from happening, the client waits for other bids during a specified interval before processing its own bid. Since contractor machines are assumed to be processing tasks for only one user at a time, the client machine's own bid is also inflated by a factor that reflects the current load on the client machine. (Human users of a processor can express their willingness to have tasks scheduled locally by setting either of these two parameters.)

"Gaming" the system

If people supply their own estimates of processing times for their tasks and these time estimates are also used to determine priority, there is a clear incentive for people to bias their processing time estimates in order to get higher priority. To counteract this incentive, the current implementation of Enterprise has an "estimation error tolerance" parameter. If a task takes significantly longer than it was estimated to take (i.e., more than the estimation error tolerance), the contractor aborts the task

and notifies the client that it was "cut off." This cutoff feature prevents the possibility of a few people or tasks monopolizing an entire system.

Conclusion

Any designer of a parallel processing computing system, whether the processors are geographically distributed or not, must solve the problem of scheduling tasks on processors. In this paper, we presented a simple heuristic method for solving this problem and analyzed its performance with simulation studies of a wide variety of situations. This scheduling heuristic is particularly suited to a decentralized implementation in which separate decisions made by a set of geographically distributed processors lead to a globally coherent schedule. Our results were encouraging about the desirability of this and similar heuristics in such a situation: (1) substantial performance improvements result from sharing tasks among processors in systems with more than light loads; (2) in many cases, these benefits are still present even when message delay times are as much as 5 to 20 percent of the average task processing time; (3) the additional benefits from pooling tasks among more than 8 or 10 machines are small; and (4) large errors in estimating task processing times cause little degradation in scheduling performance.

Finally, we described a prototype system for personal workstations on a network in which programs can easily take advantage of the maximum amount of processing power and parallelism available on a network at any time, with little extra cost when there are few extra machines available.

Acknowledgements

The first part of this work was performed while three of the authors (Malone, Fikes, and Howard) were at the Xerox Palo Alto Research Center. The work has been supported by the Xerox Palo Alto Research Center, the Xerox University Grants Program, and the Center for Information Systems Research, MIT. Portions of this paper appeared previously in Malone, Fikes, and Howard ([Mal83]).

The authors would like to thank Michael Cohen, Randy Davis, Larry Masinter, Mike Rothkopf, Vineet Singh, Henry Thompson, and Bill van Melle for helpful comments. They would also like to thank Rodney Adams and--especially--Debasis Bhaktiyar for running many of the simulations whose results are reported here.

References

[Arr60] Arrow, K., & Hurwicz, L. Decentraliztion and computation in resource allocation. In R.W. Pfouts (Ed.) *Essays in Economics and Econometrics*. Chapel Hill: University of North Carolina Press, 1960, pp. 34-104 (Reprinted in K.J. Arrow and L. Hurwicz (Eds.) *Studies in resource Allocation Processes*. Cambridge: Cambridge University Press, 1977, pp 41-95).

[Arr71] Arrow, K., & Hahn, F. *General Competitive Analysis*. San Francisco, CA: Holden Day, 1971.

[Ber82] Berhard, R. Computing at the speed limit. *IEEE Spectrum*, July 1982, 26-31.

[Bir84] Birrell, A. D., and Nelson, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 1984, 2(1), 39-59.

[Bir82] Birrell, Andrew D., Levin, Roy, Needham, Roger M., Schroeder, Michael D., Grapevine: An Excercise in Distributed Computing. *Communications of the ACM*, 25(4), April 1982.

[Bog80] Boggs, David R., Shoch, John F., Taft, Edward A., Metcalfe, Robert M., Pup: An Internetwork Architecture. *IEEE Transactions on Communications*, COM-28, (4), April 1980.

[Bri73] Brinch Hansen, P., *Operating Systems Principles*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

[Bru74] Bruno, J., Coffman, E. G., & Sethi, R. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 1974, 17, 382-387.

[Bry81] Bryant, R., & Finkel, R. A stable distributed scheduling algorithm. *Proceedings of the Second International Conference on Distributed Computer Systems*, April 1981.

[Cho79] Chow, Y. and Kohler, W. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transaction on Computers*. May 1979, C-18.

[Cof73] Coffman, Edward G., Jr., and Denning, Peter J., *Operating Systems Theory*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

[Con67] Conway, R. W., Maxwell, W. L., Miller, L. W. *Theory of Scheduling*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1967.

[Dav81a] Davis, E. and Jaffe, J. M. Algorithms for scheduling tasks on unrelated processors. *Journal of the ACM*, 1981 (October), 28, 721-736.

[Dav81b] Davies, D., Holler, E., Jensen, E., Kimbleton, S., Lampson, B., LeLann, G., Thurber, K., and Watson, R. *Distributed systems-Architecture and implementation: Lecture notes in computer science, vol. 105.* New York: Springer-Verlag, 1981.

[Dav83] Davis, R., and Smith, R. G., Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence.* Volume 20 Number 1, January 1983.

[Ens81] Enslow, P. What is a distrubted data processing system?. *Computer*, vol. 11, June 1980.

[Far72] Farber, D. J. and Larson, K. C. The structure of the distributed computing system--Software. In J. Fox (Ed.), *Proceedings of the Symposium on Computer-Communications Networks and Teletraffic*, Brooklyn, NY: Polytechnic Press, 1972, pp. 539-545.

[Far73] Farber, D., et al. The distributed computer system. *Proceedings of the 7th Annual IEEE Computer Society International Conference*, February 1973.

[Fri76] Friedman, D. & Wise, D. CONS should not evaluate its arguments. *Automata, Languages and Programming*, Edinburgh University Press, 1976, 257-284.

[Gaj85] Gajski, D aand Peir, J. Essential Issues in Multiprocessor Systems. *IEEE Computer*, June 1985, pp. 9-27.

[Gaf78] Gafarian, A. V., Ancker, C. J., & Morisaku, T. Evaluation of commonly used rules for detecting "steady state' in computer simulation. *Nav. Res. Logist. Q.*, 1978, *25*, pp. 511-529.

[Gla79] Glazebrook, K. D. Scheduling tasks with exponential service times on parallel processors. *Journal of Applied Probability*, 1979, 16, 685-689.

[Gon77] Gonzales, T., Ibarra, O. H., and Sahni, S. Bounds for LPT schedules on uniform processors, *SIAM Journal of Computing*, 1977, 6, 155-166 (as cited by [Jaffee]).

[Gra69] Graham, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 1969 (March), 17, 416-429 (summarized in Coffman and Denning, pp. 100-106.).

[Hen76] Henderson, P. & J. Morris, Jr. A lazy evaluator. *Record Third Symposium on Principles of Programming Languages*, 1976, 95-103.

[Iba77] Ibarra, O. H. and Kim, C. E. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 1977 (April), 24, 280-289.

[Jaf80] Jaffe, J. M. Efficient scheduling of tasks without full use of processor resources. *Theoretical Computer Science*, 1980, 12, 1-17.

[Jon80] Jones, A. K., and Schwarz, P. Experience Using Multiprocessor Systems - A Status Report. *Computing Surveys*, Volume 12, Number 2, June 1980.

[Kle81] Kleinrock, L., and Nilsson, A. On optimal scheduling algorithms for time-shared systems. *Journal of Association of Computing Machinery*, 28, *3*, pp. 477-486, July 1981.

[Kor82] Kornfeld, W. A. Combinatorially implosive algorithms. *Communications of the ACM*, 1982 (October), 25, 734-738.

[Kri71] Kriebel, C. H., & Mikhail, O. I. Dynamic pricing of resources in computer networks. In C. H. Kriebel, R. L. Van Horn, & J. T. Heames (Eds.), *Management Information Systems: Progress and Perspectives.* Pittsburgh: Carnegie Press, 1971, pp. 105-124.

[Kur85] Kurose, J., Schwartz, M., Yemini, Y. A microeconomic approach to decentralized optimization of channel access policies in multiaccess networks. *IEEE* , 1985.

[Lam68] Lampson, B. W. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 1968 (May), 11, 347-359.

[Lar82] Larsen, R., McEntire, P., and O'Reilly, J. *Tutorial: Distributed control.* Silver Spring, MD: IEEE Computer Society Press, 1982.

[Law79] Law, A. M., & Carson, J. S. A sequential procedure for determining the length of a steady-state simulation. *Operations Research*, 1979, *27*, pp. 1011-1025.

[Law82a] Law, A. M., & Kelton, W. D. *Simulation modeling and analysis.* New York: McGraw-Hill, 1982a.

[Law82b] Law, A. M., & Kelton, W. D. Confidence intervals for steady-state simulaitons, II: A survey of sequential procedures. 1982b *Management. Science*, vol 28, *5*, May 1982, pp. 550-562.

[Liv82] Livny, M., & Melman, M. Load balancing in homogeneous broadcast distributed systems. *Proceedings of the Computer Network Performance Symposium*, Maryland, 1982.

[Mal82] Malone, T. *A decentralized method for assigning tasks to processors.* Research meorandum, Cognitive and Instructional Sciences Group, Xerox Palo Alto Research Center, Palo Alto, Calif., August 9, 1982

[Mal83] Malone, T., Fikes, R., Howard, M., *Enterprise: A market-like task scheduler for distributed computing environments.* Working paper, Xerox Palo Alto Research Center, Palo Alto, CA, October, 1983 (Also available as CISR WP#111, Center for Information Systems Research, Massachusetts Institute of Technology, Cambridge, MA, October, 1983).

[Mal 86] Malone, T. W. and Rothkopf, M. H. Strategies for scheduling parallel processing computer systems. Paper in preparation, Sloan School of Management, MIT.

[Men85] Mendelson, H. Pricing computer services: Queueing effects. *Communications of the ACM*, 28,*3*, March 1985.

[Met76] Metcalfe, R. M., and Boggs, D. R., Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19 (7), July 1976.

[Nel81] Nelson, B. J. Remote Procedure Call. Xerox Palo Alto Research Center, CSL-81-9, May 1981.

[Nil80] Nilsson, N. J. Principles of Artificial Intelligence. Palo Alto, CA: Tioga Publishing Co., 1980.

[Ram84] Ramamritham, K. and Stankovic, J. Dynamic task scheduling in distriubted hard real-time systems. *IEEE Software*, July 1984.

[Sau81] Sauer, C. H. & Chandy, K. M. *Computer Systems Performance Modeling*. Englewood Cliffs, N.J.: Prentice Hall, 1981, pp. 296-297.

[Sch84] Schroeder, M., Birrell, A., and Needham, R. Experience With grapevine: The Growth of a Distributed System. *ACM Transaction on Computer Systems*, 1984, 2(1), 3-23.

[Sho82] Shoch, John F., Hupp, Jon A., The WORM Programs - Early Experience with a Distributed Computation. *Communications of the ACM*, 25(3), March 1982.

[Sin85] Singh, V. & Genesereth, M. A variable supply model for distributing deductions. *Proceedings of the International Joint Conference on Artificial Intelligence*, August 1985, Los Angeles, CA.

[Smi80] Smith, R. G., The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver IEEE Transactions on Computers Volume C-29 Number 12, December 1980.

[Smi81] Smith, R. G. and Davis, R., Frameworks for Cooperation in Distributed Problem Solving. *IEEE Transactions on Systems, Man, and Cybernetics*, Volume SMC-11 Number 1, January 1981.

[Sta84] Stankovic, J. and Sidhu, I. An adaptive bidding algorithm for processes, clusters and distributed groups. In *Proceedings of the Fourth Interational Conference on Distributed Computing Systems*, May 1984.

[Sta85] Stankovic, J. An application of bayesian decision theory to decentralized control of job scheduling. *IEEE Transactions of Conputers*, 1985, *C-34*, 2, pp. 117-130.

[SRC85] Stankovic, J., Ramaritham, K., and Cheng, S. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on computer*, C-34, *12*, December 1985.

[Sto77] Stone, H. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, vol Se-3, January 1977.

[Sto78] Stone, H. and Bokhari, S. Control of distributed processes. *Computer*, vol. 11, pp. 97-106, July 1978.

[Sut68] Sutherland, I. E. A futures market in computer time. *Communications of the ACM*, 1968 (June), 11, 449-451.

[Ten81a] Tenney, R., Strategies for distributed decisionmaking. *IEEE Transactions on Systems, Man and Cybernetics*, vol. SMC-11, pp. 527-538, August 1981.

[Ten81b] Tenney, R. and Sandel, Jr., N. Structures for distributed decisionmaking. *IEEE Transactions on Systems, Man and Cybernetics*, vol. SMC-11, pp. 517-527, August 1981.

[Tho83] Thompson, H. Remote Procedure Call. Unpublished documentation for Interlisp-D system, Xerox Palo Alto Research Center, Palo Alto, CA; January, 1983.

[Van81] Van der Heyden, L. Scheduling jobs with exponential processing and arrival times on identical processors so as to minimize the expected makespan. *Mathematics of Operations Research*, 1981, 6, 305-312.

[Vnt81] Van Tilborg, A. M., & Wittie, L. D. Distributed task force scheduling in multi-microcomputer networks. *Proceedings of the National Computer Conference*, 1981, 50, 283-289.

[Web82] Weber, R. R. Scheduling jobs with stochastic processing requirements on parallel machines to minimize makespan or flowtime. *Journal of Applied Probability*, 1982, 19, 167-182.

[Wit80] Wittie, L. and van Tilborg, A. MICROS, A distributed operationg system for micronet, A reconfigurable network computer. *IEEE Transactions on Conputing*, vol. C-29, December 1980.

Table 1

Mean flow times and 90% confidence intervals for the distributed scheduling protocol in various situations (different network configurations, system loads, and accuracies of processing time estimates)

90% Utilization

Network Configuration	DSP (Perfect Estimates)	DSP (± 100% Estimation Errors)	Optimal Assignment (with moving; random sequencing)
8	30.65 ± 2.27	38.96 ± 2.73	75.00
4,4	37.73 ± 2.57	46.42 ± 3.44	78.95
4,2,2	45.88 ± 2.89	53.46 ± 3.90	81.76
4,2,1,1	53.27 ± 2.98	59.75 ± 3.70	83.63
2,2,2,2	53.70 ± 3.17	60.75 ± 4.56	89.08
4,1,1,1,1	59.71 ± 2.75	67.08 ± 3.82	88.94
2,2,2,1,1	60.43 ± 3.26	67.56 ± 5.07	91.03
2,2,1,1,1,1	66.14 ± 2.91	74.23 ± 3.19	96.62
2,1,1,1,1,1,1	73.03 ± 3.14	82.46 ± 6.18	104.33
1,1,1,1,1,1,1,1	81.14 ± 2.52	90.73 ± 2.99	112.61

Table 1 (cont.)

50% Utilization

Network Configuration	DSP (Perfect Estimates)	DSP (± 100% Estimation Errors)	Optimal Assignment (with moving; random sequencing)
8	12.70 ± 0.74	14.26 ± .91	15.00
4,4	18.57 ± 1.28	20.60 ± 1.50	20.00
4,2,2	24.29 ± 1.41	24.58 ± 1.60	22.50
4,2,1,1	28.64 ± 1.84	29.68 ± 1.60	23.54
2,2,2,2	32.52 ± 1.37	32.34 ± 1.55	32.61
4,1,1,1,1	34.86 ± 1.88	36.86 ± 2.40	27.57
2,2,2,1,1	37.03 ± 1.51	36.65 ± 1.61	33.64
2,2,1,1,1,1	42.66 ± 2.35	39.04 ± 2.42	37.78
2,1,1,1,1,1,1	50.81 ± 2.34	50.88 ± 2.95	47.01
1,1,1,1,1,1,1,1	60.66 ± 2.67	61.83 ± 2.16	60.89

Table 1 (cont.)

10% Utilization

Network Configuration	DSP (Perfect Estimates)	DSP (± 100% Estimation Errors)	Optimal Assignment (with moving; random sequencing)
8	7.66 ± 0.38	8.68 ± 0.56	8.33
4,4	14.36 ± 0.66	16.15 ± 0.99	15.15
4,2,2	17.65 ± 1.06	17.37 ± 0.76	16.01
4,2,1,1	18.40 ± 1.21	17.93 ± 0.84	16.09
2,2,2,2	28.89 ± 1.21	30.36 ± 1.72	30.01
4,1,1,1,1	22.90 ± 1.27	22.06 ± 1.28	16.92
2,2,2,1,1	30.57 ± 1.45	30.12 ± 1.11	30.04
2,2,1,1,1,1	31.57 ± 1.65	33.08 ± 1.85	30.42
2,1,1,1,1,1,1	38.68 ± 2.13	40.70 ± 2.56	33.96
1,1,1,1,1,1,1,1	60.43 ± 2.60	62.99 ± 3.30	60.00

Table 2

Effect on mean flow time of adding processing capacity while keeping utilization constant (Random sequencing and SPT sequencing)

	Utilization					
	10%		50%		90%	
No. of machines	*Random*	*SPT*	*Random*	*SPT*	*Random*	*SPT*
1	66.64	61.28 ± 3.04	120.00	101.60 ± 5.92	600.00	245.20 ± 18.16
2	60.60	57.44 ± 2.64	80.00	74.28 ± 5.12	315.79	150.92 ± 10.28
4	60.01	57.78 ± 2.42	65.22	65.04 ± 2.74	178.16	107.40 ± 6.34
8	60.00	60.43 ± 2.60	60.89	60.66 ± 2.67	112.61	81.14 ± 2.52

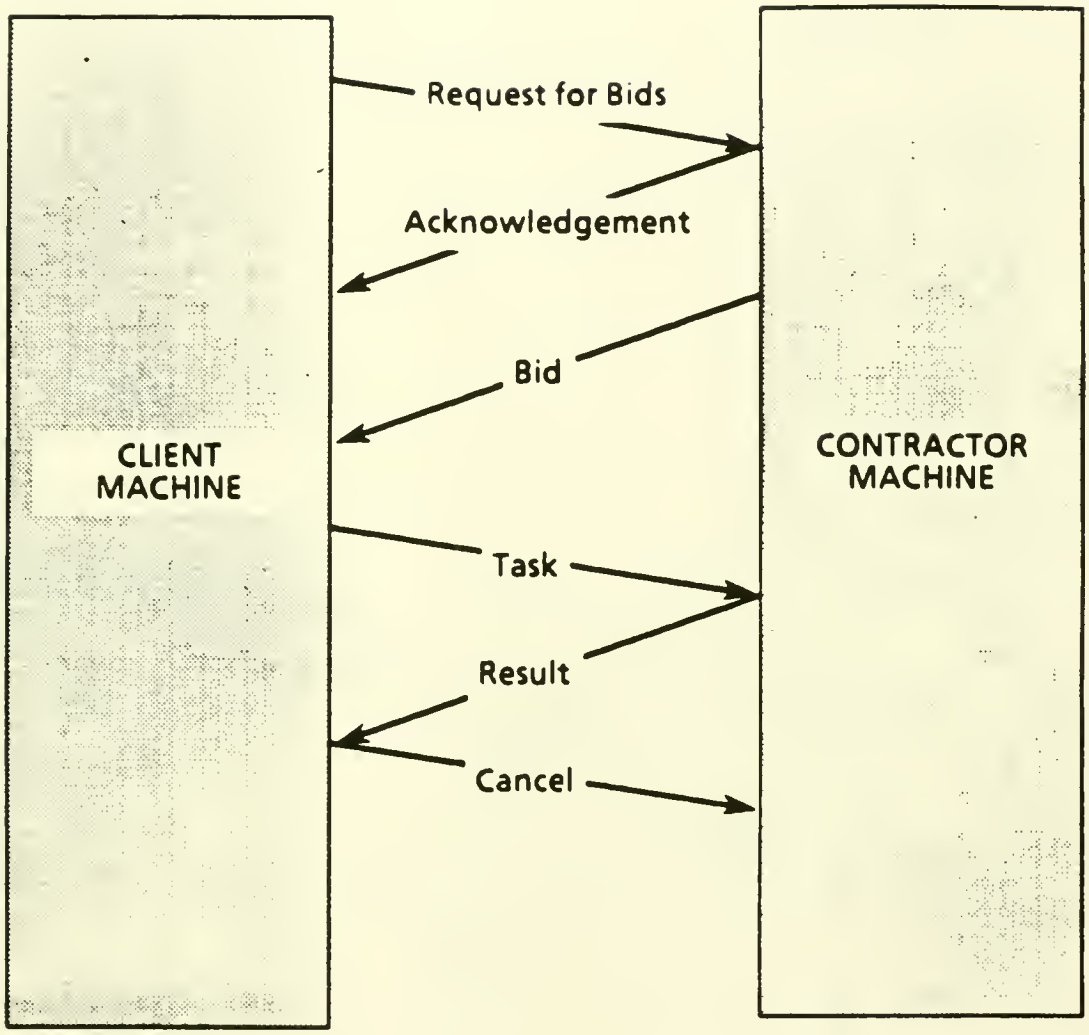


Figure 1

Messages in the Distributed Scheduling Protocol

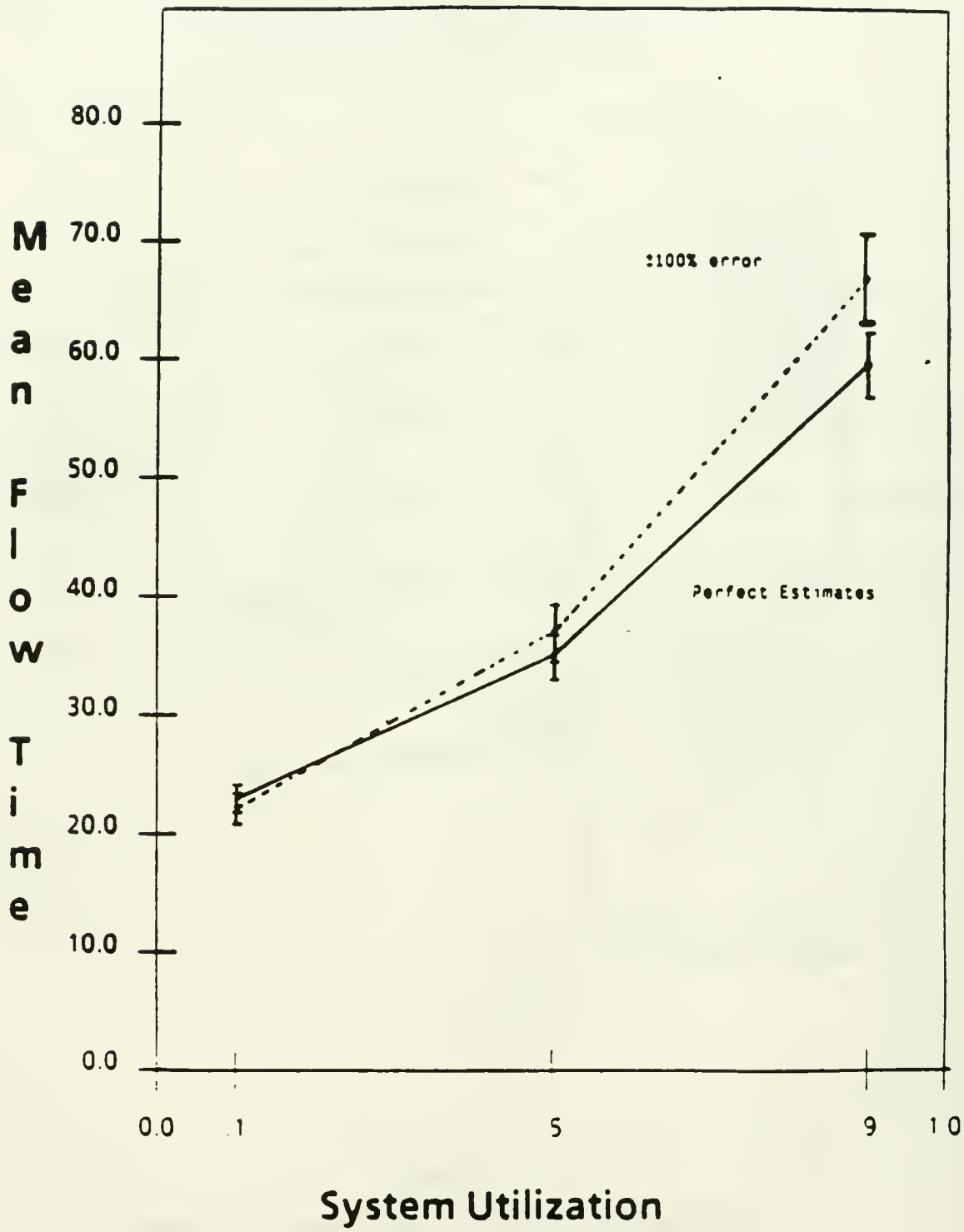


Figure 2

Effect on mean flow time of system utilization and accuracy of processing time estimates (for DSP with network configuration 41111)

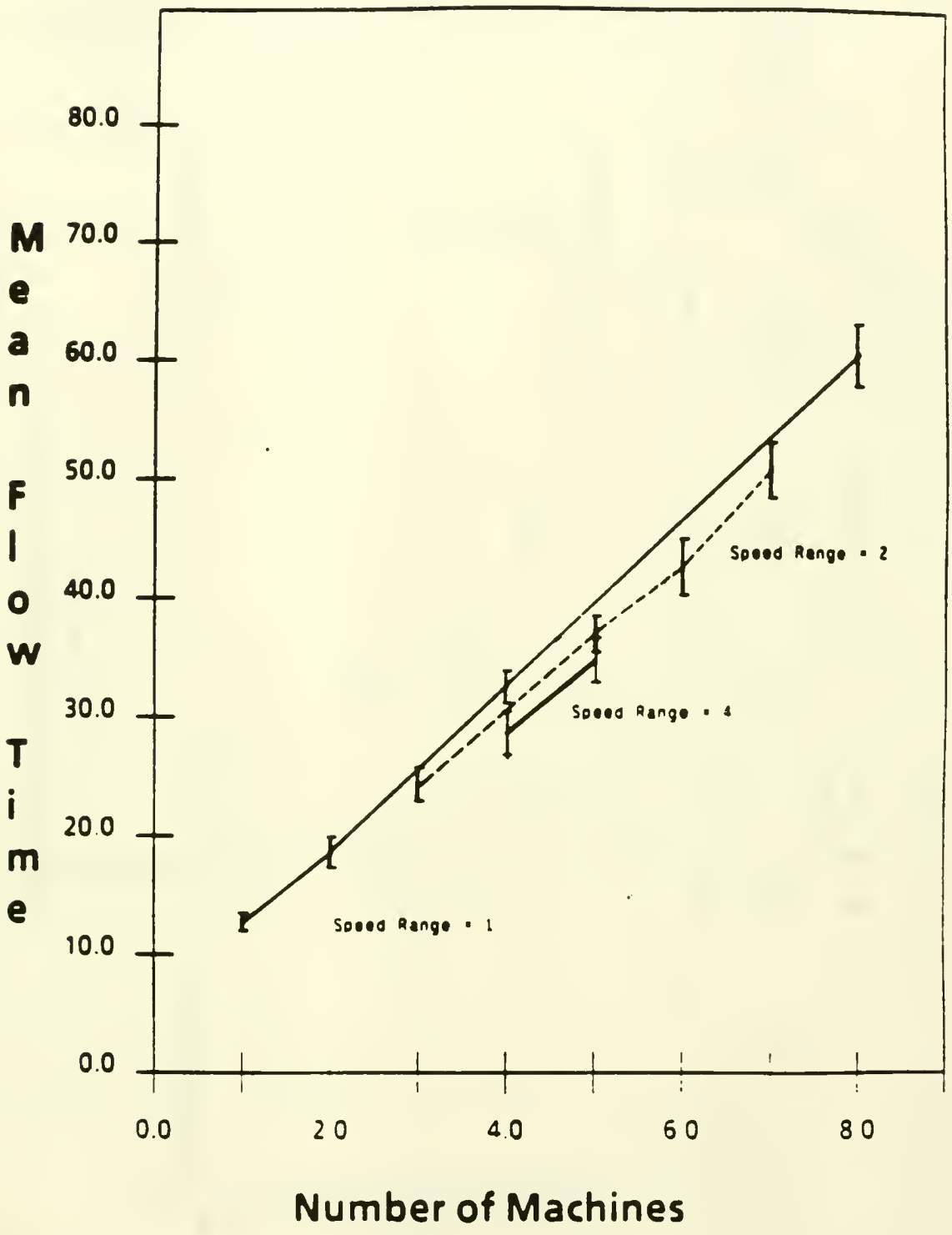


Figure 3

Effect on mean flow time of network configuration

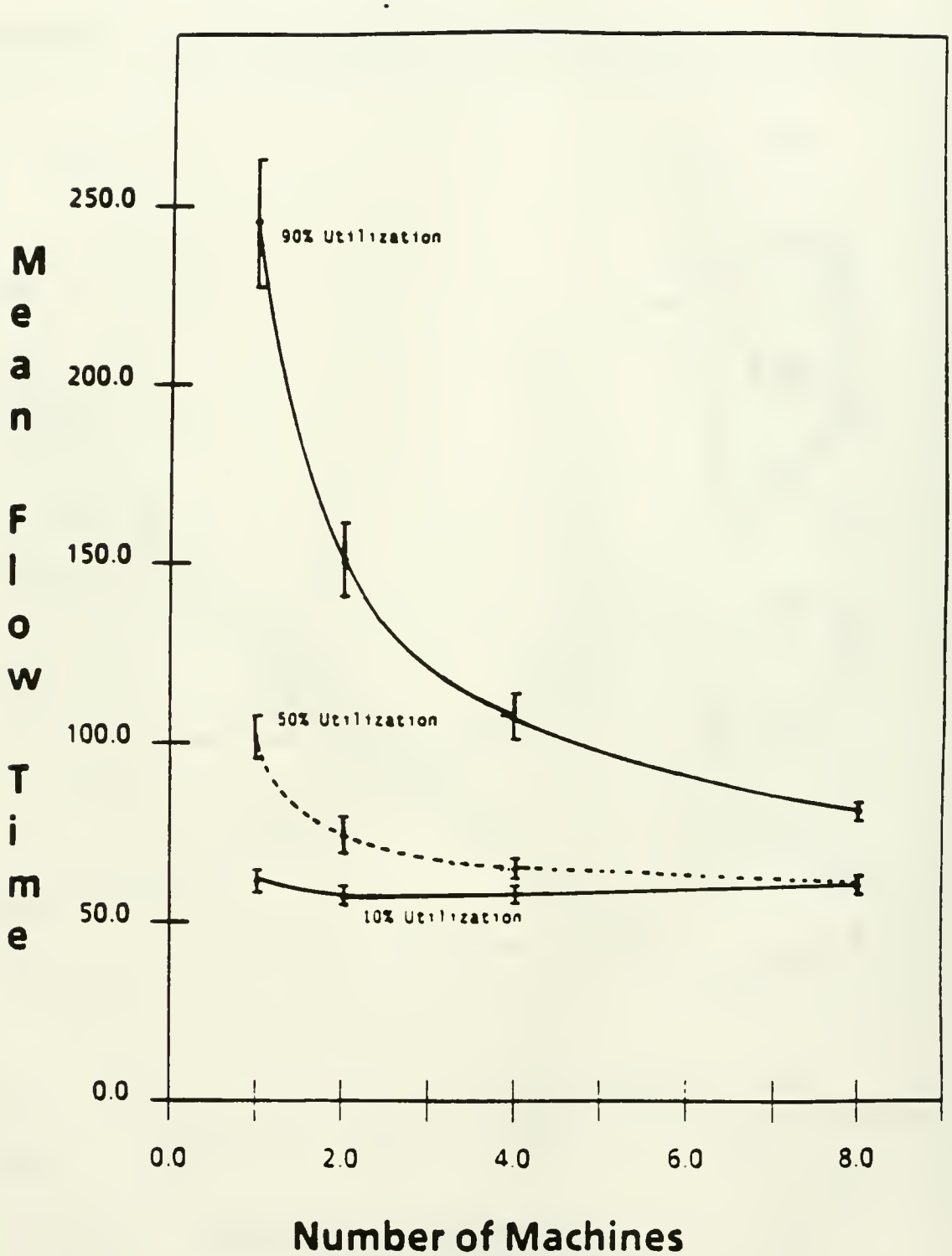
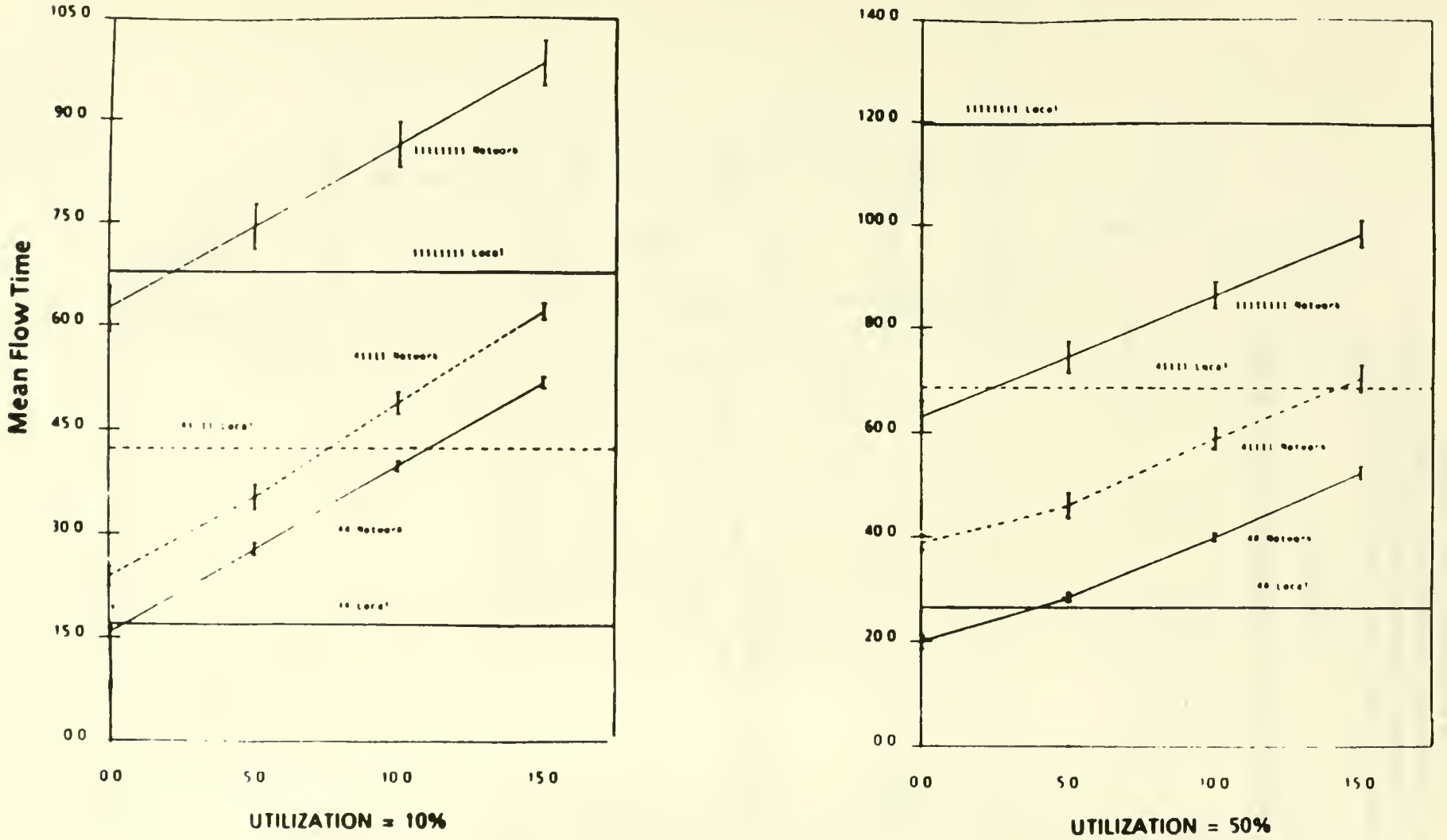


Figure 4

Effect on mean flow time of adding processing capacity
while keeping overall utilization constant
(for DSP with perfect processing time estimates
and identical machines of speed 1)



Message Delay
(Percent of average job processing time)

Figure 5

Comparison of local processing with network scheduling at various message delays, system utilizations, and configurations (DSP with perfect processing time estimates)

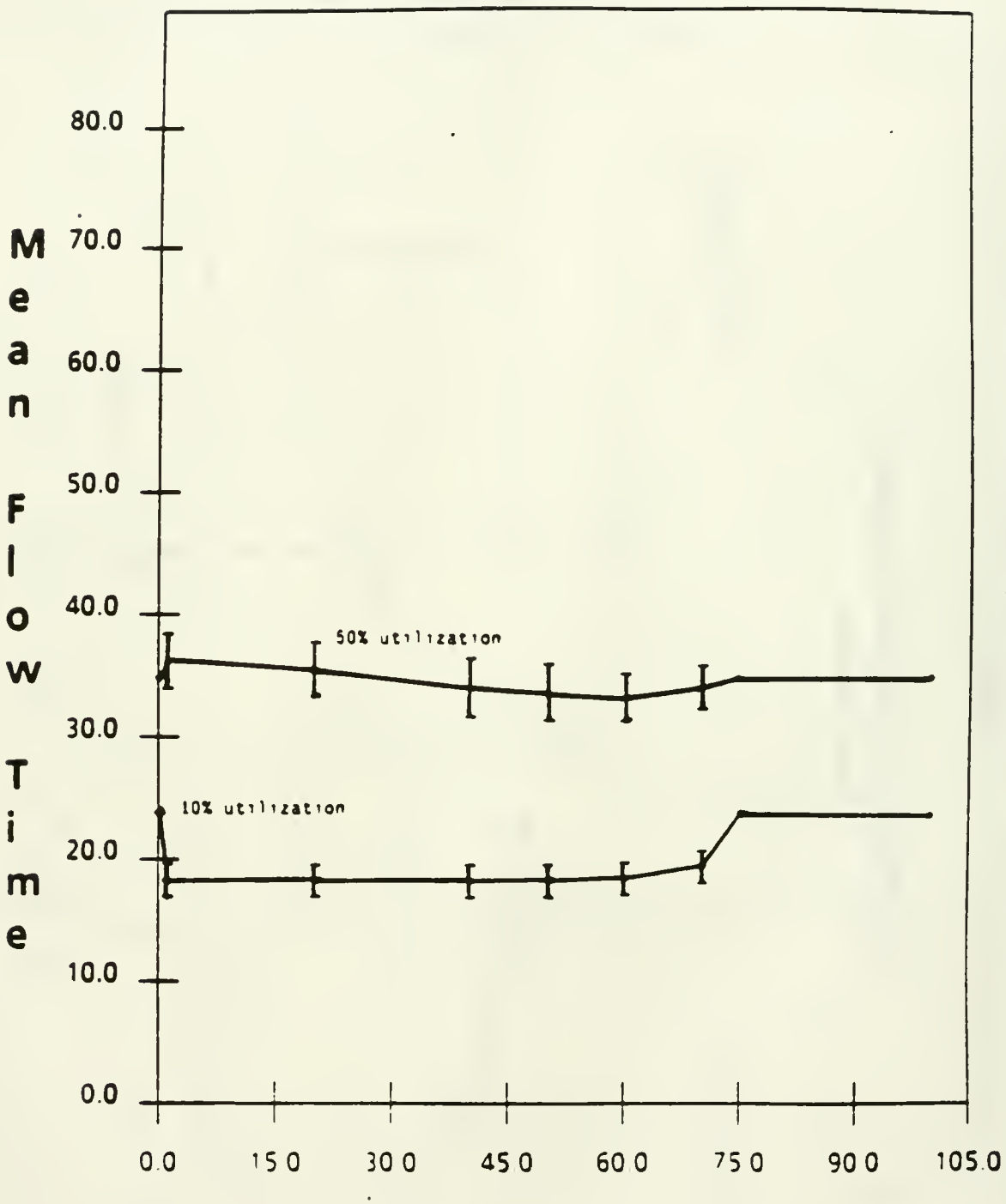


Figure 6

Effect on mean flow time of various settings
for "late bid improvement" criterion, i_0
(DSP with perfect estimates of processing time.
Network configuration 41111)

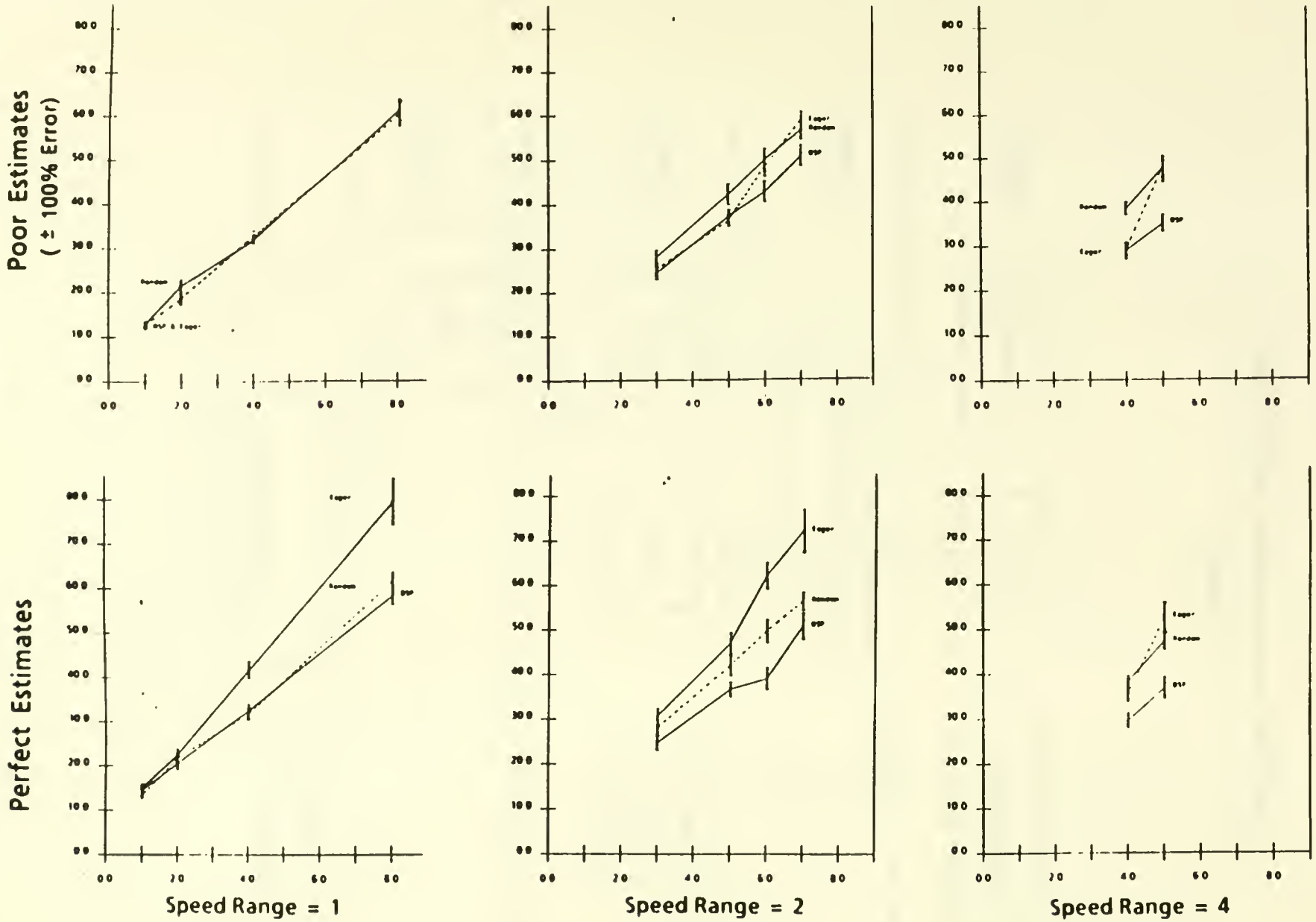


Figure 7

Comparison of DSP with eager and random alternative scheduling methods (various network configurations and accuracies of processing time estimates, system utilization = 50%)

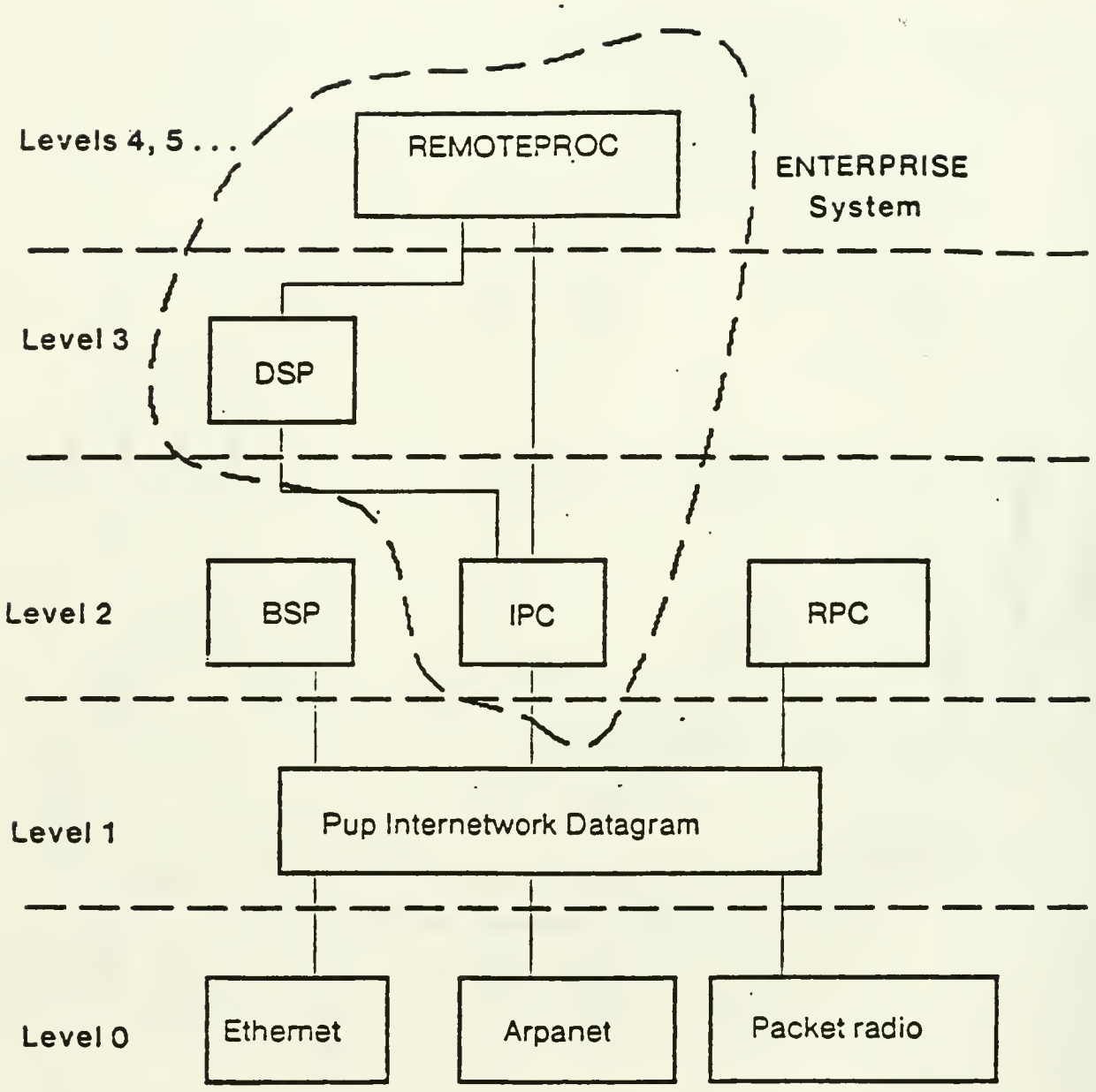


Figure 8

Protocol layers used in the Enterprise system

2384 067

BASEMENT

Date Due 2-12-90

AUG. 27 1992

MAY 3 1 2000

Lib-26-67

MIT LIBRARIES DUPL 2

3 9080 00579021 4